

KOMBINATORNO TESTIRANJE

Sadržaj lekcije

1. **Uvod i motivacija** (Problem eksplozije količine test podataka)
2. **Kombinatorne strategije** (Od pokrivanja pojedinačnih vrednosti do n-torki)
3. **Teorijske osnove konstrukcije tabela** (Ortogonalni i pokrivajući nizovi)
4. **In-Parameter-Order (IPO) algoritam** (Horizontalni i vertikalni rast)
5. **Praktičan primer: PICT alat i Cucumber** (SUT: Javalin REST API sa ograničenjima)

Motivacija: Gde nastaju greške?

- Istraživanja pokazuju da većinu softverskih otkaza izaziva **interakcija malog broja parametara**.
- **NIST (National Institute of Standards and Technology)** je kroz studije utvrdio:
 - Oko **70%** grešaka detektuje se testiranjem interakcije **2 parametra** (pairwise).
 - Preko **90%** grešaka detektuje se testiranjem interakcije **3 parametra**.
 - Retko koja greška zahteva interakciju više od **6 parametara** za njeno ispoljavanje.

Problem: Eksplozija test prostora (SUT primer)

Zamislamo jednostavan REST API endpoint implementiran u Javalin okviru za kreiranje pretplate korisnika:

// SUT: POST /subscribe

ctx.queryParam("plan"); // Vrednosti: Free, Premium, Business (3)

ctx.queryParam("payment"); // Vrednosti: CreditCard, PayPal, Wire (3)

ctx.queryParam("region"); // Vrednosti: US, EU, ASIA, Africa (4)

ctx.queryParam("device"); // Vrednosti: Web, Android, iOS (3)

Ukupan broj mogućih kombinacija (kartezijanski proizvod):

$[3 * 3 * 4 * 3 = 108]$

Šta ako sistem postane još složeniji?

- Dodajmo još samo 3 parametra u našu Javalin aplikaciju:
 `browser` (Chrome, Firefox, Safari, Edge) → 4 vrednosti
 `promoCode` (Valid, Invalid, None) → 3 vrednosti
 `currency` (USD, EUR, RSD) → 3 vrednosti
- Novi ukupan broj kombinacija iznosi:
 $[108 * 4 * 3 * 3 = 3,888]$

Zaključak: Potpuno iscrpno testiranje (*Exhaustive Testing*) postaje nemoguće u praksi zbog ograničenog vremena i budžeta.

REŠENJE: Kombinatorno testiranje

- **Definicija:** Kombinatorno testiranje je metoda crne kutije koja sistematski bira minimalan podskup kombinacija ulaznih vrednosti tako da se zadrži visok nivo pokrivenosti interakcija.
- **Cilj:** Drastično smanjiti broj test primera, a zadržati sposobnost pronalaženja grešaka izazvanih nepredviđenim ponašanjem između konfiguracija.

Kriterijumi pokrivenosti (Coverage Criteria)

U zavisnosti od željenog kvaliteta i resursa, razlikujemo sledeće strategije (od najslabije do najjače):

- 1. Each Choice (Pokrivanje svih pojedinačnih vrednosti)**
- 2. Pairwise / 2-way (Pokrivanje svih parova vrednosti)**
- 3. t-way (Pokrivanje t-torki vrednosti)**
- 4. All Combinations (Iskrpno pokrivanje)**

Strategija 1: Each Choice Pokrivanje

- **Kriterijum:** Svaka definisana vrednost svakog parametra mora se pojaviti u **bar jednom** test primeru.
- **Za naš primer:** Imamo parametre sa maksimalno 4 vrednosti (`region`).
- **Broj testova:** Može se izvesti u svega **4 test primera**.
- **Efikasnost:** Izuzetno niska. Potpuno ignoriše interakcije između parametara. Ako se aplikacija ruši samo kada je `plan=Premium` i `payment=Wire`, ovaj kriterijum to verovatno neće otkriti.

Strategija 2: Pokrivanje parova (Pairwise / 2-Way)

- **Kriterijum:** Za bilo koja dva parametra, sve moguće kombinacije njihovih vrednosti moraju biti pokrivene **bar u jednom** test primeru.
- **Zašto baš parovi?** To je industrijski “sweet spot”. Pruža najbolji odnos između broja testova i verovatnoće pronalaženja grešaka.
- Broj testova za naš inicijalni primer pada sa **108** na svega **12 do 15** test primera!

Strategija 3: Pokrivanje t-torki (t-way testing)

- **Kriterijum:** Generalizacija pairwise testiranja. Za bilo kojih t parametara, sve kombinacije njihovih vrednosti moraju biti pokrivene.
- Koristi se za visoko-rizične sisteme (avionski softver, medicinski uređaji, pametni ugovori).
- Sa porastom t , broj test primera raste polinomijalno, ali je i dalje drastično manji od ukupnog kartezijanskog proizvoda.

Teorijske osnove konstrukcije kombinatornih tabela

Kombinatorne tabele se matematički reprezentuju i generišu preko dve ključne strukture:

1. **Ortogonalni nizovi** (Orthogonal Arrays - OA)
2. **Pokrivajući nizovi** (Covering Arrays - CA)

Ortogonalni nizovi (Orthogonal Arrays)

- **Definicija:** Ortogonalni niz, u oznaci $OA(N, k, s, t)$, je matrica dimenzija $N \times k$ gde:
 - N predstavlja broj redova (test primera).
 - k predstavlja broj kolona (parametara/faktora).
 - s predstavlja broj nivoa (vrednosti po parametru).
 - t predstavlja snagu niza (*strength*).
- **Ključna osobina (balans):** U bilo kojih t kolona, svaka moguća t -torka vrednosti pojavljuje se **tačno jednak broj puta** (λ).

Primer: Ortogonalni niz $L_9(3^4)$

- Ovaj niz ima $N=9$ redova, $k=4$ parametra, gde svaki parametar ima $s=3$ vrednosti, a snaga je $t=2$.
- Uočite:* Izaberite bilo koje dve kolone. Sve kombinacije parova $((0,0), (0,1), (2,2))$ se pojavljuju **tačno jednom**.

Test	P1	P2	P3	P4
T1	0	0	0	0
T2	0	1	1	1
T3	0	2	2	2
T4	1	0	1	2
T5	1	1	2	0
T6	1	2	0	1
T7	2	0	2	1
T8	2	1	0	2
T9	2	2	1	0

Problem sa ortogonalnim nizovima u praksi

- **Strogo ograničenje:** Zahtevaju da svi parametri imaju **isti broj vrednosti** (mada postoje i *Mixed-level OA*, teže se konstruišu).
- **Prevelika balansiranost:** Zahtev da se svaki par pojavi *tačno jednak broj puta* nepotrebno povećava broj testova (N).
- U testiranju softvera nam nije važno da se par pojavi *isti* broj puta, već samo da se pojavi **bar jednom**.
- Zato u softverskoj industriji primat preuzimaju **Pokrivajući nizovi**.

Pokrivajući nizovi (Covering Arrays)

Definicija: Pokrivajući niz, u oznaci $CA(N, k, s, t)$, je matrica dimenzija $N \times k$ kod koje se u bilo kojih t kolona, svaka moguća t -torka vrednosti pojavljuje **bar jednom** ($\lambda \geq 1$).

- **Prednost:** Mnogo su fleksibilniji za realne softverske aplikacije gde jedan parametar ima 2 vrednosti, drugi 5, a treći 10 (*Mixed-level Covering Arrays - MCA*).
- **Rezultat:** Dobijaju se znatno manji skupovi testova nego kod ortogonalnih nizova.

Primer CA(11,5,3,2)

- Odgovarajući ortogonalni niz imao bi 27 redova
- Ovde ne važi svojstvo balansa

Test	P1	P2	P3	P4	P5
T1	0	0	0	0	0
T2	1	1	1	0	0
T3	2	2	2	0	0
T4	0	1	2	1	1
T5	1	2	0	1	1
T6	2	0	1	1	1
T7	0	2	1	2	2
T8	1	0	2	2	2
T9	2	1	0	2	2
T10	0	0	0	1	2
T11	1	1	1	2	1

In-parameter-order (IPO) algoritam

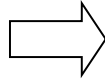
- **IPO (In-Parameter-Order)** je jedan od najpoznatijih i najefikasnijih determinističkih algoritama za generisanje pokrivajućih nizova snage t .
- Za razliku od globalnih strategija, IPO generiše testove inkrementalno, dodajući parametar po parametar.
- Za prvih t parametara konstruiše se kombinatorna tabela koja sadrži sve njihove kombinacije vrednosti.
- Ponavljaju se dve ključne faze pri dodavanju svakog novog parametra:
 1. **Horizontalni rast** (Horizontal Growth)
 2. **Vertikalni rast** (Vertical Growth)

IPO Strategija: Korak po korak

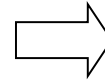
1. Primer je za snagu $t=2$ (pairwise testing). Pretpostavimo da imamo prva dva parametra P_1 i P_2 . Konstruiše se inicijalna tabela koja sadrži sve njihove parove.
2. Za svaki sledeći parametar (P_i) (od 3 do k):
 - **Horizontalni rast:** Proširi postojeće test primere dodavanjem vrednosti za novi parametar (P_i) tako da se pokrije što je moguće više novih parova.
 - **Vertikalni rast:** Ako nakon horizontalnog rasta ostanu neopokriveni parovi između novog parametra (P_i) i prethodnih parametara, dodaju se **novi redovi** (testovi) dok se svi preostali parovi ne pokriju.

Grafički prikaz rada IPO algoritma

<u>P1</u>	<u>P2</u>
1	1
1	2
2	1
2	2



<u>P1</u>	<u>P2</u>	<u>P3</u>
1	1	1
1	2	2
2	1	3
2	2	1



<u>P1</u>	<u>P2</u>	<u>P3</u>
1	1	1
1	2	2
2	1	3
2	2	1
2	1	2
1	2	3

Horizontalni rast

Vertikalni rast

- Prva dva parametra imaju po 2 vrednosti, treći ima 3 vrednosti
- Dobija se pokrivaјуći niz mešovutih nivoa

Praktičan primer primene PICT alata

- Pravimo Java aplikaciju sa REST API-jem za sistem osiguranja automobila. Endpoint prima sledeće parametre:
 - 1) **Tip Vozila (vehicleType)**: Limousine, SUV, Motorcycle
 - 2) **Starost Vozača (driverAge)**: Young (<21), Adult (21-65), Senior (>65)
 - 3) **Istorija Paketa (bonusMalus)**: Premium, Regular, Risky
 - 4) **Vrsta Osiguranja (insuranceType)**: Basic, FullCasco
- Ukupan broj iscrpnih kombinacija:
 $3 * 3 * 3 * 2 = 54$ kombinacije.

Praktičan primer – SUT aplikacija

```
public class InsuranceApp {  
    public static void main(String[] args) {  
        Javalin app = Javalin.create(config -> {  
            // Direktno registrovanje rute na router objektu  
            config.routes.post("/calculate", InsuranceApp::handleInsurance);  
        }).start(7000);  
    }  
    public static void handleInsurance(Context ctx) {  
        String vehicle = ctx.queryParam("vehicleType");  
        String age = ctx.queryParam("driverAge");  
        String bm = ctx.queryParam("bonusMalus");  
        String type = ctx.queryParam("insuranceType");  
  
        // Validacija biznis ograničenja  
        if ("Young".equals(age) && "FullCasco".equals(type)) {  
            ctx.status(400).result("Mladi vozači ne mogu uzeti FullCasco!");  
            return;  
        }  
        if ("Motorcycle".equals(vehicle) && "Premium".equals(bm)) {  
            ctx.status(400).result("Motocikli nemaju Premium status!");  
            return;  
        }  
  
        ctx.status(200).result("Kalkulacija uspešna.");  
    }  
}
```

Predstavljanje alata: Microsoft PICT

- **PICT (Pairwise Independent Combinatorial Testing)** je besplatan, open-source alat kompanije Microsoft.
- Radi preko komandne linije (CLI) i koristi napredne algoritme za generisanje pokrivajućih nizova.
- **Glavna snaga:** Podrška za **Constraints** (restrikcije/uslove), što je ključno za realni softver jer određene kombinacije parametara u stvarnosti nisu poslovno ili tehnički logične.

PICT Model Datoteka (model.txt)

Kreiramo običnu tekstualnu datoteku u kojoj definišemo parametre i njihove vrednosti:

```
vehicleType:    Limousine, SUV, Motorcycle  
driverAge:      Young, Adult, Senior  
bonusMalus:     Premium, Regular, Risky  
insuranceType:  Basic, FullCasco
```

Ako pokrenemo PICT nad ovim modelom preko komande:

```
pict model.txt
```

PICT će smanjiti broj testova sa **54** na svega **11** !

PICT Model Datoteka (model.txt)

vehicleType	driverAge	bonusMalus	insuranceType
Motorcycle	Young	Risky	Basic
Limousine	Adult	Risky	FullCasco
SUV	Adult	Premium	Basic
SUV	Senior	Regular	FullCasco
Motorcycle	Adult	Regular	FullCasco
Limousine	Senior	Premium	Basic
Limousine	Young	Premium	FullCasco
SUV	Senior	Risky	Basic
Motorcycle	Senior	Premium	Basic
SUV	Young	Regular	Basic
Limousine	Adult	Regular	FullCasco

Realnost: Uvođenje Ograničenja (Constraints)

- U realnom poslovanju osiguranja postoje striktna pravila (restrikcije):
 - 1) **Mladi vozači (Young) ne mogu uzeti FullCasco osiguranje (preveliki rizik).**
 - 2) **Motocikli (Motorcycle) ne mogu imati Premium bonus-malus status.**
- Ako ne kažemo alatu za ova ograničenja, on će generisati nevalidne test primere koje naš SUT (Javalin) odbija kroz validaciju, čime trošimo test resurse.

Prošireni PICT model sa ograničenjima

Dopunjujemo model.txt sintaksom za uslove:

```
vehicleType:    Limousine, SUV, Motorcycle
driverAge:      Young, Adult, Senior
bonusMalus:     Premium, Regular, Risky
insuranceType:  Basic, FullCasco
```

```
# Ograničenja (Constraints)
```

```
IF [driverAge] = "Young" THEN [insuranceType] <> "FullCasco";
IF [vehicleType] = "Motorcycle" THEN [bonusMalus] <> "Premium";
```

Generisani izlaz iz PICT alata

Pokretanjem `pict model.txt` dobijamo optimizovanu tabelu koja poštuje sva pravila pokrivenosti parova i sve restrikcije:

vehicleType	driverAge	bonusMalus	insuranceType
SUV	Adult	Premium	FullCasco
Motorcycle	Adult	Regular	Basic
Limousine	Senior	Risky	FullCasco
Motorcycle	Senior	Risky	Basic
SUV	Young	Regular	Basic
Limousine	Adult	Regular	Basic
Limousine	Young	Risky	Basic
SUV	Senior	Premium	Basic
Motorcycle	Adult	Risky	FullCasco
Limousine	Adult	Premium	FullCasco

Svega 10 testova umesto 54! Sva ograničenja su ispoštovana.

JUnit 5 Parametrizovani Testovi

- Alternativa BDD (Cucumber/Gherkin) pristupu ilustrovanom u primeru sa tabelama odlučivanja.
- **JUnit 5** ima izuzetnu ugrađenu podršku za data-driven testiranje preko `@ParameterizedTest`.
- Podatke generisane iz PICT alata možemo proslediti kroz `@CsvSource` ili eksterni `.csv` fajl preko `@CsvFileSource`.
- Rešenje je da izlaz iz PICT alata sačuvamo direktno u `.csv` datoteku:
 - `pict model.txt > src/test/resources/pict_izlaz.csv`

InsuranceAppTest.java (fragment)

```
@ParameterizedTest(name = "[{index}] Vozilo={0}, Starost={1}, BonusMalus={2}, Osiguranje={3}")
// Direktno čitanje iz CSV resursa, preskačemo naslovni red
@CsvFileSource(resources = "/pict_izlaz.csv", numLinesToSkip = 1, delimiter = '\\t')
public void testDozvoljeneKombinacije(String vehicleType, String driverAge, String bonusMalus,
                                       String insuranceType) {
    given()
        .queryParams("vehicleType", vehicleType)
        .queryParams("driverAge", driverAge)
        .queryParams("bonusMalus", bonusMalus)
        .queryParams("insuranceType", insuranceType)
    .when()
        .post("/calculate")
    .then()
        .statusCode(200)
        .body(equalTo("Kalkulacija uspešna."));
}
```

Izlaz testiranja u target/surefire-reports/*.xml

INFO io.javalin.Javalin -

The diagram consists of a grid of horizontal and vertical lines. A large '(-)' symbol is placed in the upper right quadrant. Several arrows point to specific points on the lines, indicating a sequence of operations or a path through the grid. The lines are labeled with various symbols, including '(-)' and '(-)'.

<https://javalin.io/documentation>

```
INFO io.javalin.Javalin - Javalin started in 322ms \o/
```

```
INFO io.javalin.Javalin - Javalin started in 222ms (0/0ms)
INFO io.javalin.Javalin - Listening on http://localhost:7001/
```

```
system-err>
```

```
stcase>
```

```

case name="testDozvoljeneKombinacije(String, String, String, String)[2] Vozilo=SUV,
=Senior, BonusMalus=Premium, Osiguranje=FullCasco" classname="app.InsuranceAppTest"
0.011"/>

```

```
case name="testDozvoljeneKombinacije(String, String, String, String)[3] Vozilo=SUV,  
t=Young, BonusMalus=Regular, Osiguranje=Basic" classname="app.InsuranceAppTest" time="0.009"/
```

```
suite>
```

Šta smo postigli ovim primerom?

1. **Ušteda vremena:** Umesto pisanja 54 ručna testa ili petlji, PICT nam je dao tačno 10 optimalnih redova.
2. **Sprečavanje lažnih rezultata:** Ograničenja u PICT-u su osigurala da Cucumber tabela ne šalje `Young + FullCasco`, što bi rezultovalo statusom 400 i oborilo naš test (koji očekuje 200).
3. **Kompletna pokrivenost parova:** Bilo koja dva parametra (npr. kombinacije `SUV` i `Risky`, `Motorcycle` i `Basic`) proverena su bar jednom kroz ove automatizovane testove.

Poređenje: Cucumber vs JUnit 5 za Kombinatorno Testiranje

Kriterijum	Cucumber (BDD)	JUnit 5 (Parametrizovani)
Čitljivost za biznis	Izuzetna (Gherkin je čitljiv netehničkim licima).	Srednja (Čitljivo samo programerima kroz kod).
Kompleksnost koda	Veća (Zahteva feature fajlove, step definitions, runner).	Minimalna (Sve je u jednoj test klasi).
Održavanje velikih tabela	Teže (Kopiranje u Gherkin tabelu).	Lakše (Direktno čitanje iz <code>.csv</code> fajla koji PICT generiše).
Izvršavanje i CI/CD	Standardno kroz Maven/Gradle Cucumber plugin.	Standardno kroz bilo koji Java test runner.

Šta je NEGATIVNO kombinatorno Testiranje?

- Do sada smo testirali samo **validne kombinacije** parametara (tzv. *Positive Testing*). Cilj je bio da osiguramo da sistem radi (vraća 200 OK) za sve ispravne parove.
- Međutim, robusni sistemi moraju adekvatno da odbiju i **nevalidne kombinacije** (one koje krše definisana *Constraints* pravila).
- **Problem:** Ako bismo ručno tražili sve nevalidne kombinacije, ponovo se suočavamo sa problemom ručnog pisanja prevelikog broja testova.
- **Rešenje:** Možemo iskoristiti **PICT alat** da za nas automatski izgeneriše i nevalidne parove koristeći tehniku **Error oznaka**.

PICT Sintaksa za Negativne Testove

PICT nam omogućava da prefiksom ~ (tilda) označimo vrednosti koje smatramo nevalidnim ili želimo da nateramo alat da generiše testove koji namerno krše restrikcije.

Ipak, najbolja praksa za biznis ograničenja (constraints) jeste korišćenje **PICT negativnog testiranja kroz restrikcije**.

Pogledajmo ponovo naša pravila:

1. `Mladi ne mogu FullCasco.`
2. `Motocikli nemaju Premium.`

Želimo matricu u kojoj će se svako od ovih kršenja pojaviti **tačno onoliko puta koliko je potrebno da se pokriju svi ostali parametri**, ali tako da se u jednom testu ne pomešaju dva različita kršenja (jer bi jedno kršenje moglo da maskira drugo).

Kako PICT rešava maskiranje grešaka?

- Ako sistem padne na prvom uslovu (npr. aplikacija odmah baci grešku jer je vozač `Young` na `FullCasco`), kod se prekida i mi nikada ne saznamo da li naša validacija za `Motorcycle` i `Premium` uopšte radi!
- To se zove **Maskiranje grešaka** (*Fault Masking*).

Generisanje Negativnih Testova preko PICT-a

Da bismo naterali PICT da izgeneriše isključivo kombinacije koje krše naša pravila, možemo kreirati poseban model `negativni_model.txt` gde obrćemo logiku restrikcija:

```
vehicleType:    Limousine, SUV, Motorcycle
driverAge:      Young, Adult, Senior
bonusMalus:     Premium, Regular, Risky
insuranceType:  Basic, FullCasco
```

Pogrešan
model!

```
# Tražimo isključivo kombinacije koje KRŠE bar jedno pravilo
Constraint [driverAge] = "Young" AND [insuranceType] = "FullCasco"
           OR [vehicleType] = "Motorcycle" AND [bonusMalus] =
           "Premium";
```

Pokretanjem ove komande dobijamo 13 test primera koji trigeruju isključivo naše 400 Bad Request validacije na backendu.

Generisanje Negativnih Testova preko PICT-a

Pokretanjem PICT sa određenim seed-om: `pict model_negativni.txt /r:587` vidimo da postoji mogućnost da se obe zabranjene kombinacije pojave u istom testu:

`vUsed seed: 587`

<code>vehicleType</code>	<code>driverAge</code>	<code>bonusMalus</code>	<code>insuranceType</code>
Motorcycle	Young	Premium	FullCasco
SUV	Young	Risky	FullCasco
Motorcycle	Young	Regular	FullCasco
. . .			

Greška je što nismo obezbedili ekskluzivno ili zabranjenih kombinacija. Pošto PICT nema operator XOR, koristimo sledeće pravilo: **$A \text{ XOR } B = (A \text{ AND NOT } B) \text{ OR } (B \text{ AND NOT } A)$**

Generisanje Negativnih Testova preko PICT-a

Ispravan negativni_model.txt generiše 13 test primera koji trigeruju isključivo 400 Bad Request validacije na backendu:

```
vehicleType:  Limousine, SUV, Motorcycle
driverAge:     Young, Adult, Senior
bonusMalus:    Premium, Regular, Risky
insuranceType: Basic, FullCasco
```

Ispravan
model!

```
# Tražimo isključivo kombinacije koje KRŠE pravila (Negacija)
# Želimo kombinacije ILI gde je vozač Young A osiguranje FullCasco ILI gde
je vozilo Motorcycle A status Premium
```

```
( [driverAge] = "Young" AND [insuranceType] = "FullCasco" AND NOT
([vehicleType] = "Motorcycle" AND [bonusMalus] = "Premium") )
OR
( [vehicleType] = "Motorcycle" AND [bonusMalus] = "Premium" AND NOT
([driverAge] = "Young" AND [insuranceType] = "FullCasco") );
```

```
@ParameterizedTest(name = "[NEGATIVAN {index}] Vozilo={0}, Starost={1}, BM={2}, Osiguranje={3}")
@CsvFileSource(resources = "/pict_izlaz_negativni.csv", numLinesToSkip = 1, delimiter = '\t')
public void testNedozvoljeneKombinacije(String vehicleType, String driverAge, String bonusMalus,
String insuranceType) {
```

```
    // Dinamički određujemo očekivanu poruku greške na osnovu podataka iz reda
    String ocekivanaPorukaGreske = "";
    if ("Young".equals(driverAge) && "FullCasco".equals(insuranceType)) {
        ocekivanaPorukaGreske = "Mladi vozači ne mogu uzeti FullCasco!";
    } else if ("Motorcycle".equals(vehicleType) && "Premium".equals(bonusMalus)) {
        ocekivanaPorukaGreske = "Motocikli nemaju Premium status!";
    }
}
```

```
given()
    .queryParam("vehicleType", vehicleType)
    .queryParam("driverAge", driverAge)
    .queryParam("bonusMalus", bonusMalus)
    .queryParam("insuranceType", insuranceType)
.when()
    .post("/calculate")
.then()
    .statusCode(400) // Očekujemo Bad Request za sve ove primere
    .body(equalTo(ocekivanaPorukaGreske));
}
```

JUnit 5 Negativni Test: Implementacija

Rezime celokupne strategije testiranja

1. **Korak 1:** Definiše se prostor ulaznih podataka (Parametri i njihove vrednosti).
2. **Korak 2:** Mapiraju se biznis restrikcije (Šta ne sme da se desi u pozitivnom scenariju).
3. **Korak 3 (PICT):** Generiše se pozitivna CSV tabela ((t=2) pokrivenost) i negativna CSV tabela.
4. **Korak 4 (JUnit 5 + Maven):** Implementiraju se dve odvojene test metode – jedna proverava stabilnost sistema (200 OK), a druga proverava validacione mehanizme aplikacije (400 Bad Request).

Ovim pristupom postizemo **maksimalan QA kvalitet sa minimalnim brojem napisanih linija koda.**

Zaključak lekcije

- **Iscrpno testiranje je nepraktično** u realnim sistemima sa mnogo konfiguracija.
- **Kombinatorno testiranje** nudi matematički dokazano rešenje za smanjenje obima testova uz zadržavanje visokog kvaliteta.
- Alati poput **PICT**-a preuzimaju na sebe posao generisanja pokrivajućih nizova.
- Integracija sa framework-ovima kao što je **JUnit** ili **Cucumber** omogućava laku i brzu automatizaciju generisanih kombinacija u Java ekosistemu.